

WEST**End of Result Set**

Generate Collection

Print

L24: Entry 1 of 1

File: USPT

Apr 8, 2003

DOCUMENT-IDENTIFIER: US 6546443 B1

TITLE: Concurrency-safe reader-writer lock with time out support

Abstract Text (1):

Synchronization services provide a concurrency-safe reader/writer lock supporting a time out feature. The lock can be implemented using lockless data structures to provide efficient synchronization services. Various features such as lock nesting and auto-transformation address common scenarios arising in componentized programs. The lock supports upgrading and suspension, and the time out feature can support an efficient, low-cost optimistic deadlock avoidance scheme. Peculiarities of the reader/writer scenario are addressed in an efficient way to maintain lock stability and consistency, thus providing synchronization services suitable for implementation at the kernel level. In one implementation using event objects, the events are managed for high efficiency and stability of the lock. For multiprocessor machines, a hybrid lock avoids a context switch by behaving as a spin lock before waiting for the lock to become available.

US Patent No. (1):

6546443

Brief Summary Text (5):

In many information processing applications, multiple executing entities attempt to access data concurrently. For example, in a database program, multiple users may attempt to access the same database tables, records, and fields at the same time. Common examples of such database programs include software for processing class registrations at a university, travel reservations, money transfers at a bank, and sales at a retail business. In these examples, the programs may update databases of class schedules, hotel reservations, account balances, product shipments, payments, or inventory for actions initiated by the individual users. Sometimes a single program executes multiple threads accessing the same data concurrently. For example, one thread may watch for changes in data made by another thread.

Brief Summary Text (7):

One would expect the value "2" to become "0" after two computers attempt to subtract "1" from it, but in the illustrated scenario, the result is instead "1." Since the algorithm failed to take concurrency into account, the database has been corrupted. Such concurrency problems can arise whenever multiple executing entities (e.g., processes, tasks, threads, processors, or programming objects) access the same data.

Brief Summary Text (13):

Second, the semaphore approach can lead to a problem called deadlock. Deadlock occurs when two or more processes (or threads) vie for two or more protected resources. For example, consider process A and process B, both of which require writing to fields Y and Z to update a database. Deadlock occurs under the following scenario: 1. Process A updates a semaphore protecting field Y to indicate Y is unavailable to other processes 2. Process B updates a semaphore protecting field Z to indicate Z is unavailable to other processes 3. Process A examines the semaphore protecting field Z and determines Z is unavailable (as noted by B), so process A waits for process B to release field Z 4. Process B examines the semaphore protecting field Y and determines Y is unavailable (as noted by A), so process B

waits for process A to release field Y 5. Both processes wait forever

Brief Summary Text (14):

Although there are ways of dealing with the deadlock problem, such as conventional deadlock detection and conventional deadlock avoidance, again, considerable computing power is typically required to implement such solutions. Also, none of the solutions completely solves the problem. In light of the difficulty of solving the deadlock problem and the relative rarity of deadlock conditions, some systems ignore the deadlock problem altogether. However, such an approach can lead to a subtle software defect that is difficult to detect and debug.

Brief Summary Text (15):

Thus, an efficient synchronization mechanism for addressing the reader/writer scenario is needed, and a mechanism for avoiding the deadlock problem is needed.

Brief Summary Text (18):

Data structures for implementing the reader/writer services can be maintained using an interlocked operation (e.g., an interlocked compare and exchange operation). Such an implementation is sometimes called "lockless" since logic to lock the data structures is not necessary. In addition, by maintaining some data structure elements in storage local to a thread, the lock services can more efficiently access lock state information.

Brief Summary Text (21):

A time out feature facilitates an optimistic deadlock avoidance scheme, providing programmers with a feature to address the deadlock problem. Various challenging programming pitfalls relating to implementing time outs are avoided.

Brief Summary Text (22):

For example, races particular to the time out arrangement are addressed to avoid lock corruption or inconsistency. Also, an event creation failure detection feature maintains stability and consistency of lock data structures in the face of insufficient available resources. The synchronization services are thus sufficiently robust for use in the kernel of an operating system or an execution engine.

Brief Summary Text (23):

In addition, the lock services support a set of features for componentized applications. For example, the services support upgrading a lock from reader status to writer status and downgrading a lock from writer status to reader status. Also, the lock can be suspended and restored. In these scenarios, information about intermediate writers (writers accessing the protected resource before the operation is complete) is provided. In addition, lock nesting can be tracked using thread-local storage, and certain nested requests can be monitored and automatically transformed to avoid deadlock.

Detailed Description Text (24):

In the following illustrated embodiments, synchronization services are provided to executing entities performing read and write operations on a protected resource. In the illustrated embodiments, the reading and writing entities are threads running in various processes; however, the illustrated principles could equally be applied to other executing entities, including processes, tasks, computer systems, processors, and programming objects.

Detailed Description Text (25):

In the illustrated embodiments, each process can have one or more threads. The practice of executing more than one thread per process is called multithreading. The illustrated embodiments thus provide useful synchronization services for use in a multithreading context, allowing programmers to more easily develop thread-safe solutions to various programming problems.

Detailed Description Text (27):

An overview of an exemplary arrangement utilizing an event-based reader/writer synchronization service system is shown in FIG. 3. In the example, a computer 304 executes a program 306. The program 306 is a client of the synchronization services 308 and comprises various objects 312 and 314 residing in a process 318, which

accomplishes work for the program 306. The threads 320 of the process 318 execute the logic associated with the objects 312 and 314, and more than one thread can be executing an object's logic concurrently. Although a single process 318 in a single program 306 is shown, there may be multiple processes and programs accessing various protected resources 330, such as a data field 332, a database table 334, or other data 336. The resources 330 are protected in that concurrent access to them is controlled to prevent corruption.

Detailed Description Text (28):

From time to time, the objects 312 or 314 require reading from and writing to (i.e., modifying) the protected resources 330. To prevent data corruption, the objects contain logic to acquire a lock before reading or writing to the protected resources 330. When acquiring the lock, the object specifies whether it will be a reader (i.e., perform only reads on the protected resource while holding the lock) or a writer (i.e., perform at least one operation modifying the resource while holding the lock). Since the threads 320 execute the logic in the objects 312 or 314, the arrangement is sometimes described in terms of reading or writing threads (or simply "readers" and "writers"). However, the synchronization services could also be used in an arrangement wherein each process has only one thread, so it may be appropriate to describe the arrangement in terms of reading or writing processes. The terms "readers" and "writers" could similarly be applied, then, to any executing entity.

Detailed Description Text (29):

In the illustrated example, an execution environment (e.g., an operating system or a virtual machine) includes a synchronization services module 308 providing lock objects 340, 342, and 344. In the example, the lock objects 340, 342, and 344 provide an interface having various methods, such as AcquireReaderLock() and ReleaseWriterLock() which are accessed by the objects 312 and 314. In keeping with the principles of object-oriented programming, the lock objects 340, 342, and 344 can include various data members for tracking the state of the lock. In one implementation, a portion of the lock state is stored in thread local storage 350.

Detailed Description Text (30):

From time to time, execution of various of the threads 320 is suspended via the synchronization services 308 using events 352 and 354. This technique is sometimes called "blocking." Typically, a thread's execution remains suspended until the event 352 or 354 is sent a resume indication, although a time out feature is supported, as described in more detail below.

Detailed Description Text (33):

The reader/writer synchronization services provide protection for a resource when properly called by executing entities. For example, a thread about to perform a read calls a "Request Reader Lock" function before performing the read. Sometimes such a sequence is called "requesting protection for a read operation," "attempting to acquire a reader lock," or "requesting a reader lock." Similarly, a thread about to perform a write is "requesting protection for a write operation," "attempting to acquire a writer lock," or "requesting a writer lock."

Detailed Description Text (34):

Typically, the acquire function returns a result code indicating success or failure (e.g., due to a time out). The sequence of requesting the lock and meeting with success is sometimes called simply "acquiring a reader (or writer) lock." Providing the protection to the requesting thread (e.g., as evidenced by providing an indication of success) is sometimes called simply "granting the lock."

Detailed Description Text (37):

Various data structures can be used to represent a lock's state. When a thread attempts to acquire a lock, the lock's state is checked and updated if appropriate. Efficiency of the lock can be increased by using an interlocked operation (e.g., interlocked compare and exchange, interlock exchange and add, interlocked increment, or interlocked test and set). For example, an interlocked compare and exchange operation can simultaneously check the lock's state and update it.

Detailed Description Text (38):

Typically, the interlocked operation provides an indication of whether the update

was successful. Failure typically indicates the lock could not be granted because the lock state could not be changed. The logic of the synchronization services may then take other steps (e.g., suspend execution of the requesting thread until the lock is available). The various interlocked operations can be incorporated in the logic of the synchronization services to avoid a separate lock protecting the lock's state.

Detailed Description Text (43):

As is described in more detail in a later section, it is common for programs to be constructed from multiple components. A single executing entity such as a thread may execute instructions in more than one component to complete work. One of the goals of object-oriented programming is to enable components from various sources to work together. Ideally, a component developer can implement logic accessing synchronization services without knowledge of the logic within clients of the component. The reader/writer lock services, however, present a challenging problem for componentized software (i.e., software composed of programming components).

Detailed Description Text (44):

Particularly, a thread may be executing a first component containing logic that acquires a writer lock. The first component might then call a second component containing logic for acquiring the same writer lock. The second component would then wait for the first component to release the lock (the lock cannot be granted to two writers), a condition that will never occur because the thread executes logic in the components sequentially. Thus, a deadlock occurs.

Detailed Description Text (45):

To prevent such a deadlock, the synchronization services provide lock nesting. Calls to the same lock by the same thread are nested by tracking a writer nest level. For example, two requests by the same thread for the same lock as a writer would result in a next level of two. Subsequently, when the thread calls a release lock function, the nest count is decremented. The lock is not actually released for use by other threads until the nest count reaches zero.

Detailed Description Text (46):

For nested readers, storage local to the executing entity (e.g., thread local storage in the case of a thread) can be used to store the nest level (e.g., an integer indicating the level of nesting). Such an arrangement can improve efficiency because only local storage need be checked to determine if the lock is available. In other words, if the thread already has been granted the lock as a reader when executing logic in a component, the lock is available for read operations by the same thread while executing another component.

Detailed Description Text (47):

Another feature called auto-transformation facilitates componentized software. For example, consider a thread executing a first component having logic acquiring a writer lock. The first component then calls a second component having logic for acquiring the same lock as a reader. At this point the thread waits for the first component to release the lock, a condition that will never occur (similar to the nesting deadlock described above).

Detailed Description Text (48):

However, it is not necessary that the second component wait because the purpose of the lock is to prevent concurrent access by a reader and a writer. In the aforementioned example, a single thread performs write and read operations serially (not concurrently). Since the thread already has the writer lock, no other thread should be performing read or write operations. Responsive to such a request, the synchronization services can transform the reader lock request into a writer lock request. Typically, the request is then nested as described above.

Detailed Description Text (55):

Under certain circumstances, in response to a lock request, the synchronization services utilize an execution suspension mechanism on which a thread waits; a resume indication can be sent to the mechanism to resume execution of the waiting thread. When the lock is granted, the synchronization services send the resume indication to the mechanism; the thread then resumes execution, having been granted the lock.

Detailed Description Text (57):

The synchronization services also support a time out for lock requests. If after a time out period expires, the request still can not be granted due to activity by other threads, the request times out, and the thread resumes execution. An indication is provided to the caller (running on a thread) that the lock was not granted (e.g., a code indicating failure). In other words, when an executing entity requests a lock, execution will resume after some waiting period, even if the lock request cannot be granted. As will be described at length, such an arrangement is useful for constructing a deadlock avoidance scheme.

Detailed Description Text (58):

One way of providing time outs is by using an execution suspension mechanism that supports time outs. After expiration of a specified time out, the execution suspension mechanism resumes execution of the suspended thread, even if no resume indication is provided to the mechanism.

Detailed Description Text (60):

Various illustrated embodiments particularly describe one execution suspension mechanism: an event object (sometimes simply called an "event"). An executing entity such as a thread can wait on the event. At some later time, execution of the thread can be resumed by sending a resume indication to the event (sometimes called "setting" the event). If a time out is provided, the thread resumes execution even if no resume indication is received. The synchronization services manage the event objects to increase efficiency and avoid corrupting a lock's state.

Detailed Description Text (61):

An event could be automatically created for each lock (e.g., an event object could be created when a lock object is created). However, an implementation can avoid undue consumption of resources by waiting until there is contention on the lock (e.g., a reader requests the lock while it is granted to a writer) before creating an event. Typically, then, the resources (i.e., those for creating events) required are bounded by the number of simultaneously-executing threads rather than the number of locks. For instance, an implementation so constructed with eighty locks and seven threads would typically require enough resources for creation of seven events, not eighty.

Detailed Description Text (69):

The following sections further describe various aspects of the reader/writer synchronization services in greater detail using various exemplary arrangements. In the following examples, the executing entities are described as threads; however, an arrangement facilitating other executing entities could be constructed in a similar fashion. Similarly, various levels of indirection (e.g., via pointers) can be added or diminished in the implementations. The synchronization services administer a lock for each protected resource having concurrent readers and writers. For purposes of the synchronization services, a group of resources may be considered a single (i.e., "logical") resource.

Detailed Description Text (71):

As described in more detail below, the reader/writer synchronization services can be crafted to support -time outs. The synchronization services can provide a timeout value as a matter of default, or a parameter can be provided with the lock request specifying a particular time out value (e.g., milliseconds). To acquire the lock with no time out a special value (e.g., -1) can be submitted as the time out value. Typically, when the thread times out, an indication is provided upon resuming execution of the thread that the thread failed to acquire the lock. Logic in the thread can thus take appropriate steps.

Detailed Description Text (76):

In addition, an event object 428 provides a suspension execution mechanism on which a thread can wait. Also, various portions of the lock data structure are stored in locations 430 local to the threads calling the synchronization services. Particularly, a reader nest level 432A, stored in a location local to a particular thread indicates the reader nesting level for the thread. Other reader levels (e.g., reader level 432N) can be stored at other locations local to other threads and

indicate the reader nesting levels for those threads. The reader nesting level indicates the number of times the reader lock has been granted to a thread. Finally, cookies 440 are used to track writer sequence number and nesting level for upgrade, downgrade, and suspend features.

Detailed Description Text (84):

The interlocked compare and exchange functionality can be used in the following way: acquire a current state of the lock in a temporary variable, modify the temporary variable to a desired value, then execute the interlocked compare and exchange functionality to swap the temporary variable into the lock (only if the lock's state has not changed). In this way, the logic avoids taking action based on a condition having been made false (by another concurrently-executing thread) before the logic completes its action. The interlocked operation can thus both check the lock state to determine if the lock is available and grant the lock by modifying the lock state together with a single instruction.

Detailed Description Text (85):

For example, the flowchart at FIG. 6 shows an exemplary method for handling a request to acquire a lock as a reader using an atomic compare and exchange. At 502, the method loads portions of the current lock state (e.g., inner state 450 of FIG. 5) into a variable. At 504, the method stores the value of the variable as an old value (for later comparison). At 506, the method checks to see if the variable indicates the lock is available for a reader. For example, as described above, a comparison of the variable against the readers mask would indicate if the lock is available for a reader. If the lock is not available, other actions 508 can be taken (e.g., wait for the lock to become available). Otherwise, at 510, one is added to the variable to increase the reader count (e.g., readers 452F) as represented in the variable. At 512, the method then attempts to exchange the variable with the now current lock state, based on whether the current lock state is still the same as the old value. The operation at 512 is performed using an interlocked compare and exchange instruction (e.g., an instruction comprising a test and set instruction native to the processor). At 514, it is determined if the instruction was successful by comparing the result of the compare and exchange with the old value (if they are equal, the compare and exchange was successful). If the instruction was successful, the lock was acquired at 516. Otherwise, the lock was not acquired at 520. An example of when the instruction would not be successful would be if a concurrently-executing thread acquired the lock as a writer during the execution of 504, 506, or 510.

Detailed Description Text (86):

The illustrated method is guaranteed to be thread-safe because a change is made to the lock state only when the assumption underlying the change (i.e., that the lock is available for a reader) remains true at the time the lock state is being changed. A method for acquiring a writer lock would work somewhat similarly; however, as more features are added to the lock, the logic becomes more abstruse. It is particularly challenging to construct synchronization service using the various interlocked operations in conjunction with supporting time outs.

Detailed Description Text (87):

Race Conditions

Detailed Description Text (88):

Developing software for concurrently executing entities (or " thread-safe" software) introduces a vexing set of problems called race conditions. An exemplary race condition arises when a first executing entity takes an action based on an assumption having since been made false by an intervening second concurrently-executing entity. For example, a first thread might check to see if there are any writing threads waiting to acquire a lock and determine that there is one waiting writer. The first thread then attempts to send a resume indication to the event on which the waiting writer is waiting. In the meantime (after the first thread determined there was a waiting thread, but before the first thread sent the resume indication), the waiting writer has timed out. Thus, the first thread has sent a resume indication to an event on which no thread is waiting (the writer has resumed execution after timing out and is no longer waiting on the event). Thus, the lock has effectively been passed to a thread no longer waiting on it. Such scenarios

may leave the system in an unstable or inconsistent state, rendering the lock unusable.

Detailed Description Text (89):

Since programmers typically write software using sequential code listings, race conditions are difficult to detect. Further, a race condition may go unnoticed because its manifestation in practice is typically very rare. And, even if the race is discovered, it may be very difficult to reproduce. Detecting and eliminating race conditions is a challenging part of developing trusted code such as that in an operating system kernel or an execution engine. Various race conditions avoided by the synchronization services are explained in more detail below.

Detailed Description Text (92):

Although the components 602, 604, and 606 might be executed simultaneously by multiple threads, each thread executes serially. Special functionality is provided in the synchronization services to support the concurrent execution of componentized software.

Detailed Description Text (95):

For example, with reference to FIG. 7, if the component 602 acquired a lock as a writer and then called component 604, which attempted to acquire the same lock as a writer, the logic might simply determine the lock is unavailable because it has already been granted. However, if the logic tracks the thread ID in the lock data structure (e.g., in writer ID 420 in FIG. 4), it can be recognized that the same thread is now requesting the lock. It is permissible to grant the same writer lock twice (or more) to the same thread because a single thread does not pose problems associated with concurrently-executing threads.

Detailed Description Text (96):

The writer nesting level of the lock is tracked (e.g., in writer nesting level 424 in FIG. 4). The nesting level is incremented upon acquisitions by the same thread and decremented upon releases by the same thread. When the writer nesting level reaches zero, the lock is actually released for use by other threads.

Detailed Description Text (97):

Similarly, a reader nesting level is tracked. However, the value can be tracked in storage local to the thread (e.g., in reader nesting level 432A in FIG. 4). In this way, the logic provides superior performance because accessing storage local to the thread improves performance compared to accessing other storage. The logic for the reader nesting level is somewhat different because the lock is not necessarily released when the nesting level reaches zero (another thread may still hold the lock as a reader).

Detailed Description Text (99):

Auto-transformation of a lock request facilitates the instance when a component 602 acquires the lock as a writer, and then calls another component 604, which attempts to acquire the lock as a reader. The logic might simply determine the lock is unavailable because it is already granted to a writer. However, by tracking the thread ID in the lock data structure (e.g., in writer ID 420 in FIG. 4), it can be recognized that the same thread is now requesting the lock. It is permissible to grant the reader lock to a thread that already holds the lock as a writer because a single thread does not pose problems associated with concurrently-executing threads.

Detailed Description Text (105):

Another aspect of the upgrade method is that the method checks to see if the thread already holds the lock as a writer or holds no lock. Thus, the upgrade method can be called to acquire a writer lock regardless of whether the thread holds the lock as a reader.

Detailed Description Text (106):

The upgrade method provides an indication of whether there were any intervening writers by observing the writer sequence (e.g., writer sequence 422 of FIG. 4). If the writer sequence has changed, then another thread intervened by acquiring the lock as a writer before the lock could be reacquired by the thread requesting the

upgrade. Under certain circumstances, intervening writers might be somewhat unlikely, so the upgrade feature would be more efficient than releasing and reacquiring the lock.

Detailed Description Text (113):

Events can function as a mechanism for synchronization between two threads because one of the threads can send a resume indication to the another thread (sometimes called "setting the event") waiting on the event. For example, a first thread can wait on an event until a second thread indicates it has finished a particular task; the second thread indicates it has finished by sending a resume indication to first thread. Thus, it can be guaranteed that the first thread will not execute until the second thread has accomplished a particular task (e.g., released the lock).

Detailed Description Text (114):

In addition, the described events support a time out mechanism. The time out allows execution to return to the thread (e.g., after a certain number of milliseconds) even if the event is not sent a resume indication. Accordingly, when the thread resumes execution after waiting on an event, the event provides an indication allowing the thread to determine whether it resumed due to a resume indication (i.e., "successfully waited") or timed out. The thread can take appropriate action depending on the outcome.

Detailed Description Text (115):

The illustrated events support a variety of operations, including creating, waiting, resuming, and releasing. In the context of a reader/writer lock, events can be used to resolve contention on the lock. In sum, a requesting thread can be forced to wait on an event if the lock is already held by another thread if granting the lock to the requesting thread would conflict with reader/writer rules (e.g., if a writing thread requests a lock already held by a reader). The event is sent a resume indication at the appropriate time (e.g., a last reader releasing the lock calls an appropriate method of the event).

Detailed Description Text (117):

The illustrated event objects support two basic event types: automatic and manual. A basic difference between the two types lies in the way each handles multiple waiting threads. When sent a resume indication, an automatic event resumes one of the threads waiting on it and then resets the event (i.e., makes the other threads wait). Accordingly, automatic events facilitate allowing threads to resume execution one at a time. When a manual event is sent a resume indication, the event resumes all of the threads waiting on the event, and the event must be reset manually by a thread. Accordingly, manual events facilitate allowing multiple threads to resume execution until the event is manually reset.

Detailed Description Text (118):

A useful analogy is to describe the event as a gate. Sending a resume indication (or "setting") the event opens the gate. Resetting the event closes the gate. For automatic events, only one thread is allowed to pass through the open gate. For manual events, the gate is simply left open for threads to pass through until the event is reset.

Detailed Description Text (121):

Various aspects of event management are handled to increase efficiency of the lock and maintain a consistent state of the lock. The features include just in time event creation, caching events, and avoiding race conditions.

Detailed Description Text (124):

An exemplary race condition encountered when managing events occurs when two writers (or two readers) race to create an event and store it in the lock state (e.g., writer event 426 in FIG. 4). Due to concurrency, the following sequence could occur when two threads, W.sub.1 and W.sub.2, attempt to acquire a writer lock. 1. The lock is held by a reader. 2. Seeing the lock is held by a reader, W.sub.1 prepares to wait on an event. W.sub.1 checks the writer event field and determines the writer event field is empty. 3. Seeing the lock is held by a reader, W.sub.2 prepares to wait on an event. W.sub.2 checks the writer event field and determines the writer event field is empty. 4. Having seen no already-existing event, W.sub.1 creates an

event E.sub.1 on which it will wait. 5. Having seen no already-existing event, W.sub.2 creates an event E.sub.2 on which it will wait. 6. W.sub.1 stores a reference to E.sub.1 in the writer event field and waits on E.sub.1. 7. W.sub.2 stores a reference to E.sub.2 in the writer event field (overwriting the reference to E.sub.1) and waits on E.sub.2. 8. A thread releasing the lock sets the event in the writer event field (E.sub.2), and W.sub.2 resumes execution. 9. W.sub.1 waits forever (or times out) because the reference to E.sub.1 has been lost.

Detailed Description Text (130):
Race Conditions Related to Time Outs

Detailed Description Text (131):
 The synchronization avoids various race conditions related to timing out threads. For example, the following sequence illustrates a possible race condition between a signaling writer and a timing out reader: 1. The lock is held by a writer. 2. A reader attempts to acquire the lock and specifies a time out; the reader is forced to wait. 3. The writer calls a release lock function. 4. The writer sees the waiting reader and determines it needs to send a resume indication to the event on which the reader is waiting. 5. The time out period elapses. 6. The reader begins a time out sequence because the time out period has elapsed; the event on which the reader was waiting is reassigned or destroyed. 7. The writer sends a resume indication to the event on which the reader was waiting (the event has since been reassigned or destroyed).

Detailed Description Text (133):
 To avoid the illustrated race condition, a signal field is provided, and logic is incorporated into the time out sequence for the acquire reader lock function. The logic checks to see if the reader was signaled (even though the time out sequence has begun). If the reader has been signaled, it simply waits on the event rather than releasing the event for reassignment or destruction. Since the event is about to be sent a resume indication, the thread simply acquires the lock instead of timing out. Preferably, a one-bit signal field is maintained for both readers and writers and resides in the lock's inner state (e.g., 450). In this way, the signal can be set in conjunction with performing other modifications to the lock's state using an interlocked operation.

Detailed Description Text (134):
 A similar condition can arise between a releasing reader and a timing out writer. By using signals, the lock state is maintained even in the face of these race conditions. The lock is thus suitable for use in kernel-level services or in services provided by an execution engine.

Detailed Description Text (135):
 Optimistic Deadlock Avoidance Using Time Outs

Detailed Description Text (136):
 As described earlier, deadlock can arise whenever at least two executing entities attempt to simultaneously hold locks on the same set of two or more resources. Table 3 shows an exemplary deadlock scenario involving two writers, W.sub.1 and W.sub.2 attempting to write to resources D.sub.1 and D.sub.2. D.sub.1 is protected by Lock L.sub.1, and D.sub.2 is protected by Lock L.sub.2.

Detailed Description Text (137):
 Each of the writers requires access to both resources to accomplish its work. However, the first writer acquires the first resource and the second writer acquires the second resource at T.sub.1; a deadlock scenario has begun. At T.sub.2, each writer attempts to acquire the resource held by the other and is forced to wait. At T.sub.3 and continuing forever at T.sub.4, each writer holds a resource required by the other and waits for a resource held by the other. Thus, neither writer can accomplish its work.

Detailed Description Text (138):
 By using time outs, the writers can implement an optimistic deadlock avoidance scheme. Table 4 illustrates a successful implementation of the scheme. Although processing begins as in the deadlock scenario of Table 3, at T.sub.5, the waiting

lock requests time out. After timing out at T.sub.7, the writers release the lock they already acquired and sleep for a random period. At T.sub.8, the first writer wakes and acquires the first lock while the second writer sleeps. Then, at T.sub.9, the first writer acquires the second lock while the second writer sleeps. At T.sub.10, the second writer finally wakes, but the first writer has already acquired the two locks. The first writer performs its work while the second writer waits. At T.sub.11, the first writer releases the locks, allowing the second writer to acquire them. At T.sub.12, the second writer can perform its work, and at T.sub.13, the second writer releases both locks. Thus, deadlock has been avoided.

Detailed Description Text (139):

The scheme is called optimistic because the logic behind the writers effectively makes an optimistic assumption that deadlock will not occur. Since the writers recover from potential deadlock situations via the timeout feature, there can be eventual progress and correctness of the lock logic is not affected. In many scenarios, deadlock is rare, and optimistic deadlock avoidance schemes tend to perform well under actual operating conditions. The logic for handling the time out condition may reside either in programming logic or the synchronization services.

Detailed Description Text (142):

The default initialization method checks the number of processors on the system. If there is more than one processor present, the default spin count is set to a predetermined value (e.g., 500). Otherwise, the default spin count is set to 0. Logic in the synchronization services consults the default spin count when a thread attempts to acquire the lock. The default spin count may vary depending upon circumstances.

Detailed Description Text (146):

The synchronization services typically provide at least four functions performing operations implemented as methods on a lock object interface (e.g., the interface 445 of lock object 404 shown in FIG. 4). A function AcquireReaderLock() acquires a lock for reading operations. According to reader/writer semantics, multiple readers may acquire the lock in this manner if there are no writers. A function ReleaseReaderLock() releases the lock for a reader. However, after one reader releases the lock, there may be other readers still holding the lock. A function AcquireWriterLock() acquires a lock for writing operations. According to reader/writer semantics, when this lock is held, no other threads (readers or writers) hold it. A function ReleaseWriterLock() releases the lock.

Detailed Description Text (149):

An exemplary implementation of an acquire reader lock method is shown in FIGS. 8A and 8B. The method can accept a desired time out value. The method could, for example, be implemented in a callable method provided by interface 445 (FIG. 4) to be called by a thread that wishes to acquire the lock represented by the lock object 404 as a reader. The flowchart of FIGS. 8A and 8B omits some logic for the sake of brevity. For example, logic for nesting readers is not shown.

Detailed Description Text (151):

If the lock is not available for a reader, the logic avoids lock operations if events are being cached at 806. For purposes of this flowchart, if either the writer and reader have been signaled (e.g., writer signaled 452D and reader signaled 452E are true), events are being cached. At 808, waiting readers (e.g., 414) is incremented. At 810 a manual event is found or created, and at 812, the logic waits on the event with the desired time out, if any. In other words, execution of the thread that requested the lock is typically suspended. In some instances, the event may have already been set (i.e., the gate is open), so execution would continue.

Detailed Description Text (153):

In either case, the logic checks to see if the instant requesting reader was the last signaled waiting reader at 830 (e.g., by checking reader signaled 408). If not, the method ends. However, if the instant requesting reader was the last signaled waiting reader, the race condition between a signaling writer and a timing out reader is avoided at 832. For example, the logic can grant the lock to the requesting reader, even though the time out sequence has begun.

Detailed Description Text (156):

An exemplary implementation of a release reader lock method is shown in FIG. 9. The method could, for example, be implemented in a callable method provided by interface 445 (FIG. 4) to be called by a thread holding the lock represented by the lock object 404 that wishes to release the lock as a reader. The flowchart omits some logic for the sake of brevity. For example, logic for de-nesting readers is not shown.

Detailed Description Text (157):

At 902, the logic checks to see if the thread holding the lock is the last reader. If not, the readers value (e.g., 406) is decremented by one at 904 to release the lock for this reader (but is still held by other readers). The decrement is accomplished using an interlocked operation (e.g., an interlocked compare and exchange on the inner lock state 450).

Detailed Description Text (158):

If the thread holding the lock is the last reader, the logic checks to see if there are any waiting writers at 906. If so, an automatic event is found or created at 908 (avoiding the race of two concurrently-executing entities trying to create an event). Then, the writer signaled value (e.g., 412) is turned on at 910, and the readers (e.g., 406) is decremented by one at 912 to release the lock. 910 and 912 are accomplished using an interlocked operation. Then a waiting writer is resumed (via the event) at 920.

Detailed Description Text (159):

If there were no waiting writers, the logic checks if there were any waiting readers at 922. If so, a manual event is found or created at 924 (avoiding the race of two concurrently-executing entities trying to create an event). The reader signaled value (e.g., 408) is turned on at 926 and the readers value (e.g., 406) is decremented by one at 928 to release the lock. 926 and 928 are performed using an interlocked operation. Then the waiting readers are resumed (via the event) at 930.

Detailed Description Text (162):

An exemplary implementation of an acquire reader lock method is shown in FIG. 10. The method can accept a desired time out value. The method could, for example, be implemented in a callable method provided by interface 445 (FIG. 4) to be called by a thread that wishes to acquire the lock represented by the lock object 404 as a reader. The flowchart of FIG. 10 omits some logic for the sake of brevity. For example, logic for nesting writers is not-shown.

Detailed Description Text (164):

If the lock was not available, the waiting writers value (e.g., 416) is incremented at 1008. An automatic event is found or created at 1010, and the thread then waits on the event with the specified time out, if any at 1012. Upon resurning, the logic checks to see if the wait timed out at 1020. If not, the waiting writers value (e.g., 416) is decremented and the writer value (e.g., 410) is set to one at 1022 with an interlocked instruction (e.g., to modify the inner state 450).

Detailed Description Text (165):

If the wait timed out, the waiting writers value (e.g., 416) is decremented at 1030. And a time out race condition is avoided at 1032 (e.g., by checking writer signaled 412). Specifically, the race condition is similar to that described for readers at 832. For example, the logic can grant the lock to the requesting writer, even though a time out sequence has begun.

Detailed Description Text (167):

An exemplary implementation of a release writer lock method is shown in FIG. 11. The method could, for example, be implemented in a callable method provided by interface 445 (FIG. 4) to be called by a thread holding the lock represented by the lock object 404 that wishes to release the lock as a reader. The flowchart omits some logic for the sake of brevity. For example, logic for de-nesting writers is not shown.

Detailed Description Text (168):

At 1104, the logic checks if there are any waiting readers. If so, a manual event is

found or created at 1110 (avoiding the race of two concurrently-executing entities trying to create an event). At 1112, the reader signal (e.g., reader signaled 408) is turned on. At 1114, the writer (e.g., writer 410) is cleared. 1112 and 1114 are performed together with an interlocked instruction. Execution then branches to box 1132.

Detailed Description Text (169):

If there were no waiting readers, the logic checks if there are any waiting writers at 1120. If so, an automatic event is found or created at 1122 (avoiding the race of two concurrently-executing entities trying to create an event). At 1124, the reader signal is turned on. At 1126, the writer is cleared. 1124 and 1126 are performed together with an interlocked instruction. Execution then branches to box 1132.

Detailed Description Text (172):

An alternate way of describing a release writer method is shown at FIG. 12. Initially, the lock's inner state (e.g., 450) can be read into a temporary variable; the temporary variable is saved in an old value variable for later comparison when updating the lock's state. At 1202, notation is made that there will be no writer, since this writer is releasing the lock. The notation (and other notations indicated in the later-described steps) is made by modifying the temporary variable. At 1204, the logic checks if there are any waiting readers. If so, a reader event is found or created at 1210 (avoiding the race of two concurrently-executing entities trying to create an event) and a notation is made that the waiting reader is being signaled at 1212.

Detailed Description Text (173):

If there were no waiting readers, the logic checks if there are any waiting writers at 1220. If so, an event is found or created at 1222 (avoiding the race of two concurrently-executing entities trying to create an event), and a notation is made that the waiting writer is being signaled at 1224.

Detailed Description Text (177):

At 1302 (13A), notation is made that there will be one less reader, since this reader is releasing the lock. The notation (and other notations indicated in the later-described steps) is made by modifying the temporary variable. At 1304, the logic checks if the reader is the last reader (i.e., if this is the only thread holding the lock). If not, the logic continues at 1342 as described below. If so, the logic checks to see if there are any waiting writers at 1306. If so, an automatic event is found or created at 908 (avoiding the race of two concurrently-executing entities trying to create an event). Then, a notation is made at 1310 that a writer is being signaled, and the logic continues at 1342 as described below.

Detailed Description Text (178):

If there are no waiting writers, the logic checks if there are any readers waiting on the lock at 1322. If so, a manual event is found or created at 1324 (avoiding the race of two concurrently-executing entities trying to create an event). A notation is made at 1326 that a reader is being signaled.

Detailed Description Text (186):

The source code listing included in the computer program listing appendix file "source-txt" is an exemplary implementation of concurrency-safe reader/writer synchronization i.e., services supporting timeouts. In the exemplary implementation a linked-list of lock structures (i.e., RWLocks) the thread has acquired is maintained in the thread local storage. The exemplary implementation reuses the thread local storage data structures efficiently and increases size only when needed. The exemplary implementation is able to check for the fast path case (e.g., when acquiring the lock as a reader) by examining the thread-local storage.

Detailed Description Paragraph Table (1):

TABLE 1 Lock Fields stored by a Lock Object Name Function Readers 406 Indicates the number of readers holding the lock Reader Signaled 408 Indicates when the lock is being passed to a waiting reader; can serve as a communication mechanism between a thread passing the lock and a thread that has resumed execution due to a timed out event; the resuming thread can acquire the lock to avoid a corrupt lock state. This

field can be used to avoid race conditions. It can also be used to indicate when event caching operations are being performed. Writer 410 Indicates whether a writer holds the lock Writer Signaled 412 Indicates when the lock is being passed to a waiting writer; can serve as a communication mechanism between a thread passing the lock and a thread that has resumed execution due to a timed out event; the resuming thread can acquire the lock to avoid a corrupt lock state. This field can be used to avoid race conditions. It can also be used to indicate when event caching operations are being performed. Waiting Readers 414 Indicates the number of readers waiting on the lock Waiting Writers 416 Indicates the number of writers waiting (requested but not yet acquired) on the lock Writer ID 420 Stores the thread ID of the thread currently holding the lock Writer Sequence 422 Incremented each time the lock is granted to a writer Writer Nesting Level Number of times the writer lock has been granted 424 to the thread holding the lock as a writer Writer Event 426 Points to an event on which waiting writers can wait for the lock Reader Event 427 Points to an event on which waiting readers can wait for the lock

Detailed Description Paragraph Table (3):

TABLE 3 Deadlock Scenario W.sub.1 's action W.sub.1 's action W.sub.2 's action W.sub.2 's action Time on D.sub.1 on D.sub.2 on D.sub.1 on D.sub.2 T.sub.1 request and request and acquire acquire T.sub.2 hold request and request and hold wait wait T.sub.3 hold wait wait hold T.sub.4 hold forever wait forever wait forever hold forever

Detailed Description Paragraph Table (4):

TABLE 4 Optimistic Deadlock Avoidance Success Scenario W.sub.1 's action W.sub.1 's action W.sub.2 's action W.sub.2 's action Time on L.sub.1 on L.sub.2 on L.sub.1 on L.sub.2 T.sub.1 request and request and acquire acquire T.sub.2 hold request and request and hold wait wait T.sub.3 hold wait wait hold T.sub.4 hold wait wait hold T.sub.5 hold time out time out hold T.sub.6 release none none release T.sub.7 sleep sleep sleep T.sub.8 request and none sleep sleep acquire T.sub.9 hold request and sleep sleep acquire T.sub.10 write to write to request and none resource resource wait T.sub.11 release release acquire request and acquire T.sub.12 write to write to resource resource T.sub.13 release release

Other Reference Publication (10):

Kleiman et al., Programming with Threads, SunSoft Press, pp. 248-253, 1996, pp. 259-260, pp. 273-274.

Other Reference Publication (11):

Lewis et al., Threads Primer: A Guide to Multithreaded Programming, SunSoft Press, pp. 65-72, 1996, pp. 87-96, p. 117, p. 205, pp. 224-227.

Other Reference Publication (26):

Frost, "Portable Thread Synchronization Using C++," Software Tool & Die, <http://world.std.com/.about.jimf/papers/c++sync/c++sync.html>, pp. 1-6, Jun. 7, 1999.

Other Reference Publication (38):

"Threads Support Code," <http://goya.inescn.pt/.about.avs/dotNoweb/support/current/threads.red.html>, pp. 1-11, Jun. 1, 1999.

Other Reference Publication (41):

Birrell, "An Introduction to Programming with Threads," Digital Systems Research Center, pp. 1-33, Jan. 6, 1989.

CLAIMS:

5. The method of claim 1 wherein the first executing entity is a first thread executing in a process, and the second executing entity is a second thread executing in the process.

25. The method of claim 24 wherein the lock is a first lock and the requesting executing entity holds a second lock, the method further comprising: responsive to the indication that the request has timed out, releasing the second lock to avoid a

deadlock condition.

33. The method of claim 31 wherein the first executing entity is a thread and the reader nesting count resides in thread local storage of the thread.

34. The method of claim 31 wherein the first executing entity is a thread associated with a thread local storage, and the thread local storage stores a thread local data structure having a reference to a data structure representing the lock and a reader nesting count, the method further comprising: checking the thread local data structure to determine if a fast path is available to acquire the lock as a reader; and responsive to determining the fast path is available, taking the fast path to acquire the lock as a reader.

48. In a computer system having a plurality of executing threads and a plurality of protected resources, a method of avoiding deadlock in a deadlock avoidance scheme, the method comprising: tracking how many threads are reading each protected resource and whether a thread is writing to each protected resource in locks having per-lock data structures; whenever a first thread is unable to acquire a first lock protecting a first resource due to contention on the first lock, blocking the first thread on an event, specifying a time out period; if contention on the lock has not ceased after the time out period, timing out on the event to unblock the first thread; after unblocking the first thread, releasing, with the first thread, a lock on a second protected resource; and waiting for a sleep period to allow a thread other than the unblocked thread to access the first protected resource and the second protected resource, thereby avoiding deadlock.

49. The method of claim 48 further comprising: performing an interlocked operation on the data structure of the first lock to grant the first lock to the first thread.

50. In a computer system, a method of providing reader/writer synchronization services to a set of a plurality of threads via a lock, the method comprising: allowing a plurality of the threads to simultaneously hold the lock as a reader; and preventing any of the of the plurality of the threads from holding the lock as a writer while any other of the plurality of the threads holds the lock as a reader, wherein said preventing any of the of the plurality of the threads from holding the lock as a writer while any other of the plurality of the threads holds the lock as a reader comprises observing failure of an interlocked operation; and preventing any of the plurality of the threads to hold the lock as a writer while any other of the plurality of the entities holds the lock as a writer, wherein said preventing any of the plurality of the threads to hold the lock as a writer while any other of the plurality of the entities holds the lock as a writer comprises observing failure of an interlocked operation.

51. The method of claim 50 further comprising: resuming execution of a thread waiting on the lock after a time out period has expired.

52. The method of claim 51 further comprising: providing an indication to the thread waiting on the lock that a time out period has expired.

63. In a computer system, a lock object for providing reader/writer synchronization services to a set of a plurality of components executing on a thread, the lock object comprising: a readers field for tracking how many executing entities out of the set currently hold the lock as a reader; a writer field indicating whether an executing entity out of the set currently holds the lock as a writer; a callable method for receiving a request from a first component out of the set to hold the lock as a reader, the callable method comprising performing an interlocked operation on the readers field and the writers field to grant the request to hold the lock as a reader, wherein the callable method for receiving a request to hold the lock as a reader maintains a reader nest level field for each thread, wherein the reader nest level tracks how many unreleased grants of the lock have been made to a thread as a reader, the callable method operable for receiving a request to hold the lock as a reader from a component called by the first component and responsive to said call to increase the reader nest level; and a callable method for receiving a request to hold the lock as a writer, the callable method comprising performing an interlocked

operation on the writer field and the reader field to grant the request to hold the lock as a writer, wherein the callable method for receiving a request to hold the lock as a writer maintains a writer nest level, wherein the writer nest level tracks how many unreleased grants of the lock have been made to a thread as a writer.

64. The lock object of claim 63 wherein the reader nest level for each thread is maintained at storage local to each thread.